*Модел. и анализ информ. систем.* Т. **17**, № **4** (2010) 17–26

УДК 004.41

# Генерация тестовых данных на основе формального анализа данных конфигурации проекта

Батаев А. В., Давыдов А. А., Налютин Н. Ю., Синицын С. В.[1]

*Национальный исследовательский ядерный университет «МИФИ»*

*e-mail: dron02@iskratelecom.ru, bataev@cyber.mephi.ru, nnalutin@gmail.com, svsinitsyn@mephi.ru*

*получена 20 ноября 2010*

**Ключевые слова:** технология программирования, управление разработкой, функциональное тестирование, тестовые данные, логические уравнения, управление конфигурациями

Предлагается метод подготовки тестовых данных, обеспечивающих определенный уровень покрытия требований для функционального тестирования. Использование метода упрощает поддержание в непротиворечивом состоянии конфигурации данных жизненного цикла проекта, включающей в себя требования, программный код и сгенерированные тесты. Вводится классификация дефектов программного обеспечения. Предлагается подход к формализации анализа требований и реализации тестируемой системы, основанный на представлении разбиения на классы эквивалентности в виде системы логических уравнений. Предложен приблизительный метод решения получаемых уравнений. Обсуждается применимость подхода в реальных процессах промышленных проектов.

## 1. Введение

Одной из важнейших компонент жизненного цикла современного программного обеспечения является верификация. Одним из методов верификации программного обеспечения является функциональное тестирование. Основное назначение функциональных тестов – выявить несоответствия между реализацией и функциональными требованиями. Комбинированное тестирование, включающее функциональное тестирование и элементы структурного тестирования, позволяет выявлять не только функциональные несоответствия, но и неисполнимые команды, недостижимые участки кода и т.д. [1].

В условиях длительных программных проектов применяются автоматические тесты. Разработка тестовых процедур выделилась в отдельную область деятельности программистов, интегрированную в общий процесс разработки программного обеспечения.

[1]Научно-исследовательская работа проводится в рамках реализации ФЦП “Научные и научно-педагогические кадры инновационной России” на 2009–2013 годы, конкурс НК-437П, государственный контракт П2616 от 26.11.2009

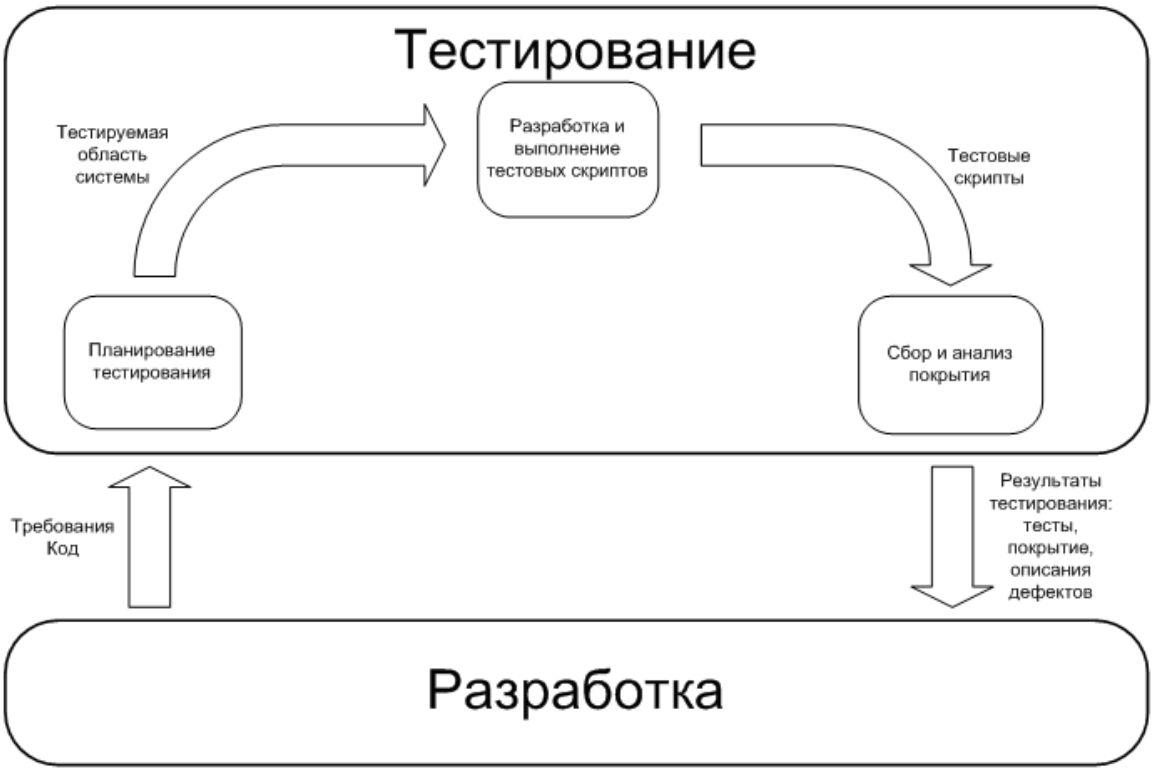


Рис. 1. Этапы процесса тестирования

Процесс тестирования сложных систем можно разделить на несколько этапов (Рис. 1): a) планирование тестирования, то есть определение, какие тесты для каких компонент системы нужны; b) разработка и выполнение тестов, то есть формирование тестовых данных, подготовка автоматических тестов и исследование системы с их помощью, с целью поиска проблем; c) сбор и анализ покрытия кода, то есть анализ того, какие части кода системы были выполнены в процессе тестирования, для оценки полноты тестирования и поиска неисполнимых и неописанных частей.

На этапе планирования необходимо определить, какие тесты нужно создать, обновить и выполнить на очередной версии целевой системы. Для этого необходимо отслеживать соответствие между тестовыми сценариями, требованиями и компонентами системы и обеспечивать их согласованность на всех этапах жизненного цикла программного проекта.

Поддержка требований, программного кода компонентов системы и тестов в согласованном состоянии является одной из задач технологического процесса управления конфигурациями [2, 3]. В ходе разработки высококритичных программных систем основная функция процесса управления конфигурациями заключается в предотвращении неконтролируемого развития проекта и обеспечении гарантии того, что все изменения, вносимые в проект, учитываются и санкционируются в соответствии с принятой технологией разработки. В состав процессов управления конфигурациями входит идентификация данных жизненного цикла разработки, управление изменениями, а также обеспечение трассируемости между объектами конфигурации

и контроль их статуса.

Основной сложностью в подготовке тестов является построение достаточного, но при этом предельно малого набора тестовых сценариев. Использование такого набора сценариев позволяет сократить время тестирования, сохранив требуемый уровень качества. В то же время такой набор сценариев дает возможность минимизировать количество объектов конфигурации проекта и сократить время выявления несоответствий между объектами. При этом необходимо заметить, что интерпретация критериев достаточности набора тестов представляет собой отдельную задачу, решаемую в рамках каждого конкретного проекта.

В статье предлагается метод, который может помочь в решении нескольких задач тестирования, таких, как установление соответствия между кодом и требованиями и генерация тестовых данных для обнаружения расхождений между кодом и требованиями, а также для выявления участков кода, которые не описаны в требованиях.

Предлагаемый подход основан, в том числе, на анализе требований и, соответственно, применим только при достаточной степени их формализма. Например, при работе со схемами Simulink или логическими схемами MatLab. В случае неформализованных требований его применение требует разработки формальной модели на их основе.

## 2. Постановка задачи

Введем некоторые обозначения с целью определить требования, выдвигаемые к набору тестовых данных.

Будем рассматривать программу как отображение $S : D \to R$, где $D$ – область определения программы, а $R$ – область значений. Требования к программе $S$ определяются множеством $OR \subseteq Dr \times Rr$, где $Dr$ – область определения программы на основании требований, а $Rr$ – область значений на основании требований. В этом случае для некоторого вектора входных значений $d \in D$ значение программы $S$ при заданных входных значениях $d$, то есть $S(d) \in R$, будет считаться «правильным», то есть соответствующим требованиям, если $d \in Dr$, $S(d) \in Rr$, а пара $\langle d, S(d)\rangle \in OR$. Таким образом, программа $S$ считается «правильной», если она дает правильный результат на любом входном векторе, то есть $\forall d \in D \Rightarrow (d \in Dr)\&(S(d) \in Rr)\&(\langle d, S(d)\rangle \in OR)$.

«Неправильной» будем называть такую программу, для которой $\exists d \in Dr : \langle d, S(d)\rangle \notin OR$. Такая ситуация говорит о том, что в программе существует дефект. Его проявлением является отказ программного обеспечения. Отказ есть возврат «неправильного» выходного значения.

Учитывая особенности исходного кода для многих критических систем (а именно, отсутствие динамических конструкций, циклов, внутренним состоянием которых можно пренебречь), можно абстрагироваться от внутреннего состояния программы. Например, рассматривать состояние как один из компонентов входного и выходного вектора или рассматривать эквивалентную программу, лишенную внутренних состояний $Sn : Dn \to Rn$, то есть анализировать последовательность из $n$ последо-

вательных воздействий и реакций исходной программы.

В более общем случае дефекты в программах можно классифицировать на основе входных значений и соответствующих им выходных значений:

1) завершение программы в результате обработки входных значений, определенных в требованиях, но которые некорректно обрабатываются программой;

2) несоответствие между значениями, возвращаемыми программой, и значениями, ожидаемыми на основании требований;

3) обработка входных значений, не предусмотренных требованиями, и выработка на их основе выходных значений.

Используя введенные ранее определения, можем выразить данную классификацию следующим образом (Рис. 2):

для класса 1 – $\exists d \in Dr : (d \notin D)$;

для класса 2 – $\exists d \in Dr : (d \in D) \& (\langle d, S(d)\rangle \notin OR)$;

для класса 3 – $\exists d \in D : (d \notin Dr)$.

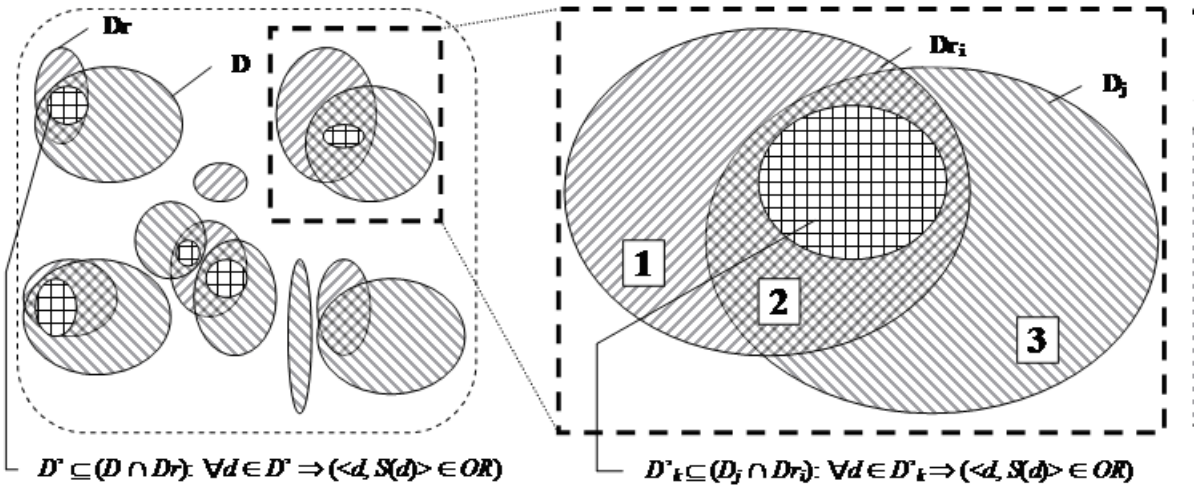


Рис. 2. Классы дефектов на основе областей определения

Обозначим области, на которых проявляются дефекты:

для класса 1 – $D1 = \{d|((d \in Dr) \& (d \notin D))\}$;

для класса 2 – $D2 = \{d|((d \in Dr) \& (d \in D) \& (\langle d, S(d)\rangle \notin OR))\}$;

для класса 3 – $D3 = \{d|((d \notin Dr) \& (d \in D))\}$.

Области входных переменных можно рассматривать как совокупность непересекающихся множеств, т. е. область $D \equiv \cup_j D_j$, а область $Dr \equiv \cup_i Dr_i$. На пересечении этих областей значений входных переменных существует еще одна область $D' \equiv D \cap Dr$, в которой поведение программы соответствует требованиям. Соответственно, эта область тоже рассматривается как совокупность фрагментов, то есть $D' = \cup_k D_k$ , где $D'_k \subseteq (Dj \cap Dri) : \forall d \in D'_k \Rightarrow (\langle d, S(d)\rangle \in OR)$. В программе существуют дефекты второго класса, если $\exists d{\in}Dr : (d{\in}D) \& (d{\notin}D')$.

Отдельным тестом будем называть любой вектор входных значений $t{\in}(D{\cup}Dr)$. Отдельный тест t считается пройденным, если значение $S(t)$ является «правильным»: $\langle t, S(t)\rangle{\in}OR$. В этом случае тест $T$, являющийся множеством отдельных те-

стов $t$, есть конечное подмножество множества $(D \cup Dr) : T \subset (D \cup Dr)$. Тест $T$ считается успешным, если все отдельные тесты $t$, составляющие тест $T$, являются успешными: $\forall t \in T \Rightarrow \langle t, S(t) \rangle \in OR$.

Основная цель тестирования состоит в том, чтобы обнаружить отказ программы и тем самым выявить в ней дефект. Тест $T$ назовем результативным, если хотя бы один отдельный тест, входящий в данный тест, не является пройденным, т.е. $T$ результативен, если $\exists t \in T : S(t)$ «неправильно». Другими словами, тестовое воздействие вызывает отказ программного обеспечения одного из 3-х возможных типов: $\exists t \in T : (t \in D1 \cup D2 \cup D3)$. Соответственно, задача подготовки тестовых данных – это задача построения набора $T$, выявляющего, в идеальном случае, все дефекты.

## 3. Суть подхода

Предлагается метод генерации результативных тестов на основе разбиения области определения программы на несколько подобластей в соответствии с принципом эквивалентности [1, 2, 3]. Суть метода состоит в том, что входные данные программы разбиваются на несколько классов, в рамках которых данные обладают некоторыми общими свойствами, определяющими сходное поведение системы. В предлагаемом подходе такие области выделяются не только для области $Dr$, определяемой требованиями к программному обеспечению, но и для области $D$, определяемой кодом программного обеспечения.

Для области $Dr$ такая общность выражается в виде единичного требования (или его части), применимого для всех входных значений, принадлежащих одной области эквивалентности. Для области $D$ общность определяется следующим образом: все входные значения относятся к одной области эквивалентности, если при задании этих значений программа выполняется по одному и тому же пути или они приводят к одному и тому же результату.

Если программа полностью соответствует требованиям, то области определения будут совпадать. Также будет совпадать с этими областями и область $D'$. В общем случае можно говорить о том, что каждая подобласть одной области является подобластью другой области: $\forall Dr_i((Dr_i \subseteq D) \& (Dr_i \subseteq D'))$ и $\forall D_j((D_j \subseteq Dr) \& (D_j \subseteq D'))$.

Таким образом, выделив все области эквивалентности на основе требований к программному обеспечению, все области на основе всех допустимых путей в программе и сравнив их, можно сделать вывод о том, совпадают ли данные области. Если эти области не совпадают (см. рис. 2), то можно говорить о том, что программа не соответствует требованиям, и, следовательно, в программе имеются дефекты. Точки, принадлежащие разности областей, дадут результативные тесты. В том случае, если программа соответствует требованиям, методом не будет сгенерировано ни одного теста.

## 4. Метод генерации тестовых данных

Метод генерации тестовых данных основан на попытке описать множества $D1$, $D2$ и $D3$ на основе функций, позволяющих определить принадлежность определенных

входных значений к тому или иному множеству.

1. На основе формальных требований к программному обеспечению строятся предикаты $p_{in}^{req}$ и $p_{out}^{req}$, описывающие пред- и постусловия к программному обеспечению. При этом множество $Dr$ полностью определяется предикатом $p_{in}^{req}$. Итоговая область должна быть разбита на непересекающиеся подобласти, и соотношение $p_{in}^{req}$ представляет собой выражение вида $\vee_i p_{ini}^{req}$, где каждое соотношение $p_{ini}^{req}$ описывает атомарную подобласть $Dr_i$.

2. Граф программы анализируется в соответствии с методами вывода предусловий из постусловий, начиная с конечной вершины. Предполагая, что программа завершается при отсутствии каких-либо дополнительных условий, можем считать, что все переменные имеют значения, не выходящее за пределы границы ассоциированных типов и физических ограничений разрядной сетки. В этом случае образуется предикат вида: $p_{out}^{true} \equiv \wedge(xi \in [d_{mini}, d_{maxi}])$, где $x_i$ – все используемые переменные программы, а $d_{min}$ и $d_{max}$ – границы, определяемые типами переменных. Затем, применяя правила вывода предусловий из постусловий, выводится предусловие для программы.

Назовем подпутем в программе элементарный неделимый путь в некотором составном пути. Каждый путь в программе представляет собой совокупность элементарных путей. Их количество и состав однозначно определяется наличием в ограничениях условий пути операций ИЛИ. Например, в программе if (A1 or A2) then B1 else B2 (A1 и A2 – простые логические переменные, а B1 и B2 – некоторые блоки кода) существует два пути (A1 or A2) -> B1 и not(A1 or A2)->B2, но в первом пути существует два элементарных подпути (A1)->B1 и (A2)->B1, а во втором пути – всего один подпуть (not A1 and not A2)->B2.

Полученное предусловие представляется в виде совершенной дизъюнктивной формы. В результате получается система условий, наложенных на все входные переменные, при которых программа или участок кода выполняется: $\vee_{mn} p_{mn}^{true}$, где $p_{mn}^{true}$ – ограничение, $m$ – индекс пути в программе, а $n$ – индекс подпути в заданном пути.

3. Строится соотношение, определяющее область $D'$. Для этого в качестве постусловия выбирается выражение $p_{out}^{req}$, на основе которого по коду строится выражение $p_{in}^{req'}$. Данное выражение и описывает область определения $D'$, в которой поведение кода совпадает с требованиями.

4. На основе полученных соотношений выделим области $D1$, $D2$ и $D3$:

D1: $\vee(p_{ini}^{req} \wedge not(p_{ini}^{true}))$;

D2: $\vee(p_{ini}^{req} \wedge p_{mn} \wedge not(p_{in}^{req'}))$;

D3: $\vee(p_{mn} \wedge not(p_{in}^{req}))$.

5. Необходимо проанализировать каждый дизъюнкт итоговых выражений, описывающих области $D1$, $D2$ и $D3$. Для каждого дизъюнкта необходимо определить хотя бы одно значение входных переменных, при котором рассматриваемый дизъюнкт становится истинным. То есть найти хоть одно решение соответствующего уравнения. Если такое значение существует, то это значит, что в программе существует дефект, приводящий к отказу. Значение входных переменных, при которых подусловие принимает истинное значение, берется в качестве отдельного теста $t$, позволяющего выявить дефект в коде тестируемой программы.

Каждый дизъюнкт в выражениях, описывающих области $D1$, $D2$ и $D3$, соответ-

ствует некоторому элементарному пути в программе (подпути), который приводит к отказу.

Любое решение уравнений для предикатов, описывающих классы эквивалентности на множестве $Dr$, дает набор тестовых данных, покрывающих область определения программы в соответствии с требованиями. При этом каждый полученный тест однозначно определяется соответствующим уравнением и однозначно трассируется на требование к системе, описываемое этим уравнением.

В случае, если описания требований достаточно формальны (т.е. представляют собой систему соотношений в терминах логики первого порядка), а код программы написан с учетом некоторых ограничений (программа не содержит циклов, сложных операций над массивами и указателями), то возможна полная автоматизация процесса генерации тестовых данных. В некоторых случаях можно предложить замену процесса функционального тестирования предлагаемым методом, как средством обнаружения дефектов.

В общем случае формирование логических уравнений – достаточно сложная задача, требующая привлечения человека. В такой ситуации при подготовке тестовых данных неизбежно возникает задача описания предикатов классов эквивалентности, что, в той или иной мере, все равно приходится решать в процессе функционального тестирования. При этом необходимо разрабатывать тесты, используя полученные данные, и проверять реальное поведение программы на всех классах эквивалентности.

Решение задачи формирования логических уравнений позволит сократить затраты и на повторное тестирование, т.к. исполнение автоматических тестов в целях регрессионного тестирования значительно проще и дешевле повторного анализа всей системы. То есть на очередной итерации анализу подвергаться может только определенная часть системы, в которой процессами конфигурационного управления были выявлены изменения.

# 5. Алгоритм разрешения логических выражений

Основная проблема анализа итоговых систем ограничений состоит в дальнейшем их разрешении. Для целей дальнейшего разрешения итоговых систем ограничений имеет смысл использовать приблизительные методы решения.

Сведем исходную задачу к задаче минимизации функции $P'(X)$, такой, что если точка $X_0 : P(X_0) = True$, то:

- $P'(X_0) = 0$.
- $P'(X) > 0$ для всех $X \neq X_0$.
- $\{lim_{X \to X_0}(gradP')(X) \to \overrightarrow{0}$, то есть $X_0$ является стационарной точкой функции $P'(X)$.

Сделаем аппроксимацию функции в окрестности стационарных точек, исходя из предположения, что между отдельными точками нет разрывов либо параметры представляют собой значения с плавающей запятой. Исходя из данного предположения $(gradP')(X)$ представляет собой вектор такой, что

$(gradP')(X) = \{lim_{\Delta x_1 \to 0} \frac{\Delta P'(x_1)}{\Delta x_1}, lim_{\Delta x_2 \to 0} \frac{\Delta P'(x_2)}{\Delta x_2}, ..., lim_{\Delta x_n \to 0} \frac{\Delta P'(x_n)}{\Delta x_n}\}$

Если же уравнение $P(X) = True$ имеет решения не для одной точки $X_0$, а для некоторого пространства точек $S_0$, то описанные выше ограничения должны выполняться для всех точек $X_{0k} \in S_0$.

В качестве $P'(X)$ можно использовать модификацию оригинальной функции $P(X)$, в которой все элементарные логические функции и все функции, возвращающие логический результат, заменяются арифметическими операциями. Эти операции заменяются в соответствии с правилами, описанными в Таблице 1.

Функция $distance(x, y)$ оценивает расстояние между двумя значениями и может принимать различные формы. Для числовых значений она может быть задана как функция вида $distance(x, y) = min(abs(x - y) + emin, emax)$, где $emax$ – наибольшее допустимое значение для типа переменных, а $emin$ – ближайшее к 0 значение данного типа.

Таблица 1. Преобразование логических функций

| Оригинальная функция | Арифметический аналог |
|---|---|
| c = a rel b, где rel - $>, <, \geq, \leq, =, \neq$ | c'.Straight = if(a rel b) then 0.0 else distance(a, b)<br>c'.Not = if(a rel b) then distance(a, b) else 0.0 |
| c = a op b, где op – and или or | c'.Straight = if(a op b) then 0.0<br>else (a'.Straight + b'.Straight)<br>c'.Not = if(a op b) then (a'.Not + b'.Not) else 0.0 |
| c = not(a) | c'.Straight = a'.Not, c'.Not = a'.Straight |
| True | c'.Straight = 0, c'.Not = 1 |
| False | c'.Straight = 1, c'.Not = 0 |

Таким образом рассчитываются не только значения логических выражений, но одновременно и их отрицание. Поле Straight используется для расчета значений выражений, а поле Not – одновременно для расчета отрицания тех же самых выражений. Это позволяет облегчить вычисление функций отрицания и избавляет от дополнительного разложения логических выражений для приведения их в более удобную для анализа форму. Результат решения содержится в итоговом поле Straight.

Смысл функции $P'(X)$ заключается в том, что она позволяет оценить «удаленность» текущего значения False функции $P(X)$ от ближайшего достижимого значения True. В этом смысле, операция поиска минимума функции $P'(X)$ сводится к попытке минимизировать расстояние между значениями False и True.

Использование эмпирических методов поиска минимума функций не гарантирует получения решений. Поэтому процесс поиска решений функций может разбиваться на несколько этапов. Те функции, для которых были найдены решения, исключаются из списка нерешенных и не подвергаются дальнейшему анализу. Оставшиеся функции повторно анализируются с уточненными параметрами процедуры поиска, позволяющими проводить более глубокий анализ функций.

# 6. Заключение

Предложена классификация дефектов программного обеспечения и метод формализации подготовки тестовых данных. Предлагаемый способ позволяет при некоторых условиях, то есть при наличии требований, которые можно привести к соотношениям в форме логики первого порядка, и кода, в котором не используются циклические конструкции, сложные конструкции с указателями и массивами, не только автоматизировать подготовку тестовых данных, но и использовать его для выявления дефектов вместо тестов.

В прочих случаях построение систем логических уравнений – это задача, требующая для своего решения привлечения человека. Решение подобной задачи может требовать значительных усилий. Однако использование метода формализует процесс подготовки тестовых данных и определения достаточности подготовленных данных. Кроме того, обеспечивается трассируемость тестов на элементарные части системы. Таким образом повышается управляемость и контролируемость процесса тестирования.

В случаях ручного анализа необходимо реализовывать автоматические тесты на основании полученной информации, чтобы избежать решения этой задачи в полном объеме в дальнейшем развитии тестируемой системы.

Предложенный подход был положен в основу метода, позволяющего частично автоматизировать процесс тестирования. Метод был практически опробован в процессе заказного тестирования программного обеспечения системы управления полетами FMS (Flight Management System).

# Список литературы

1. Синицын С.В., Налютин Н.Ю. Верификация программного обеспечения. М.: Бином; Лаборатория знаний Интуит, 2008. 368 с.

2. ГОСТ Р 51904-2002. Программное обеспечение встроенных систем. Общие требования к разработке и документированию. М.: Госстандарт России, 2002. 94 с.

3. Налютин Н.Ю., Синицын С.В. Проблемы управления конфигурациями в процессе разработки программного обеспечения встроенных систем // Программные продукты и системы. 2008. № 1. С. 26–29.

4. Соммервилл И. Инженерия программного обеспечения. М.: Вильямс, 2002. 624 с.

5. Hayhurst K. J., Veerhusen D.S. et al. A practical tutorial on Modified Condition / Decision Coverage: Technical Memorandum. NASA, 2001. 85 p.

# Test data generation based on a formal analysis of the project configuration

Bataev A.V., Davydov A.A., Nalutin N.Y., Sinitsyn S.V.

**Keywords:** software engineering, software management, functional testing, test data, logic equation, configuration management

This article discusses a problem of test data preparation for functional testing with a defined level of test coverage. Application of the method simplifies the management of software project configuration, keeping requirements, code and tests in a consistent state. Classification of software deffects is presented in the article. An approach the formalizing the target system code and requirements analysis is proposed. This method is based on the representation of equivalence class partitioning as logical equations. An original method to get equations solutions is also provided. Method applicability in real industrial projects is discussed.

**Сведения об авторах:**
**Батаев Алексей Владимирович**,
Национальный исследовательский ядерный университет «МИФИ»,
канд. техн. наук, старший преподаватель;
**Давыдов Андрей Александрович**,
Национальный исследовательский ядерный университет «МИФИ»,
аспирант, инженер;
**Налютин Никита Юрьевич**,
Национальный исследовательский ядерный университет «МИФИ»,
канд. техн. наук, старший преподаватель;
**Синицын Сергей Владимирович**,
Национальный исследовательский ядерный университет «МИФИ»,
канд. техн. наук, доцент;